Petri Nets for Component-Based Software Systems Development

Leandro Dias da Silva[1], Kyller Gorgônio [2] and Angelo Perkusich[2]
[1]Paraiba State University, [2]Federal University of Campina Grande
Brazil

1. Introduction

The Software Engineering discipline was created to try to apply techniques and methods of others engineering disciplines to software systems development. To achieve this goal it was necessary to change the way software was developed, not only at code level, but also at the process level. Like in other engineering disciplines, one of the major objectives of software engineering is to develop artifacts in a systematic way. Several building block approaches were proposed and developed along the years. Nowadays one of the most researched and used approach are software components (Crnkovic and Grunske, 2007. Nierstrasz et al., 2002). Components are autonomous units with independent life cycle that represent an specific functionality. A component consists of functionality, interface and possibly other non functional characteristics.

The development of bigger systems with components as building blocks is called Component Based Development (CBD). To make this possible it is necessary to adapt the traditional software engineering techniques and methods, or even defined new ones, to attend to specific CBD requirements. In the context of Component Based Software Engineering (CBSE) the objective is to define a set of practices that promotes the CBD.

Formal methods improve the development process of software and hardware systems by helping designers to achieve dependability at different levels of abstractions such as requirements, specification, modeling and design. This is mainly due to the fact that the application of formal methods helps discovering and removing errors by performing automatic analysis and verification (Clarke and Wing, 1996). Petri nets (Murata, 1989), and more specifically Hierarchical Coloured Petri Nets (HCPN) (Jensen, 1992. Jensen, 1997) are a very powerful tool that has been widely studied and applied for the specification and analysis of complex concurrent systems (Donatelli and Thiagarajan, 2006. Kleijn and Yakovlev, 2007. Jensen, 2005. Jensen, 2006). It has a graphical representation that helps the design of complex software systems. There are several advantages of using a formal method in systems design such as, automatic simulation, proof of properties and unambiguous documentation.

In the context of software engineering, the reuse of artifacts in the development of new software systems increases the productivity. Also, the reuse of artifacts that are well known to be correct is an effective way to increase the dependability on the system under development. Reuse is not restricted to pieces of source code, but it can be also be applied to requirements, documentation, project decisions and specifications. During the last few years, component based software engineering has been applied to promote the development of software systems based on reuse (Szyperski, 1999. Crnkovic , 2001. Crnkovic et al., 2002. Crnkovic and Grunske, 2007. Nierstrasz et al., 1992. van Steen et al., 1998).

The basic premise during the reuse process is that the designer should observe that in specific application domains, different software systems share some common characteristics. These characteristics can be represented by any kind of artifact, such as source code or a model described using some formal language, HCPN in the context of this chapter. Therefore, the identification of such common characteristics is a very important task. Firstly, when an artifact that has been already modeled is identified, it is possible to search for it in a repository and reuse it with some adaptations, instead of modeling it from scratch. Secondly, after an artifact is modeled and verified, it can be made available to be reused in the future.

The main goal of this chapter is to introduce a systematic and automatic approach to the reuse of HCPN models in the specification and verification of complex software systems. The focus is on the study and development of techniques that help the automation of the modeling phase, reducing time and money costs of the project. This approach is an alternative to *ad-hoc* reuse practices in which the reuse process is of the entire responsibility of the developer. In order to achieve this goal the approach for the specification and analysis of components, frameworks for components composition, and component-based software systems is presented. The proposed approach is guided by a reuse process and software tools for automatic manipulation of the models. Moreover, a case study is used to illustrate its application. The work presented in this chapter is based on the use of temporal logic (Emerson, 1990), model checking (Clarke et al., 1999. McMillan, 1993) and supervisory control theory (Ramadge & Wonham, 1989) in order to support: adaptation, integration and use verification of HCPN models. It is fully implemented in CPN/ML language (Christensen & Haagh, 1996) for a well known set of tools for HCPN models called Design/CPN (Des, 2006). From the application point of view, the introduced approach is used to develop models in the context of complex embedded software systems. Embedded systems have been applied in several kinds of computing devices (Nierstrasz et al., 2002) such as automobiles, cellular phones and control and automation devices. Due to the evolution of the technology, more complex devices executing more complex tasks are being developed, making it difficult to deal with the increasing complexity from the software point of view (Lee, 1999. Lee, 2002). As discussed by Knight (Knight, 2001. Knight, 2002. Knight, 2004), two major problems that must be tackled in this domain are specification and verification. The first one is mainly related to the need to build models that are more dependable. The later one is related to the difficulty in performing tests on embedded software systems. Therefore techniques such as model checking can help to early detect design errors. The approach introduced in this chapter is an effective approach to deal with these two problems.

The rest of the chapter is organized as follows. In section 2 the basics of the formal tools used in the present work are introduced, including a discussion about Coloured Petri nets, temporal logic and model checking. The application of reuse techniques to build formal models is discussed in Section 3. In Section 4 an example of an embedded system to illustrate the reuse process is presented. The adaptation, integration, and use verification steps are described in Sections 5, 6, and 7, respectively. Finally, in Section 8 the chapter is concluded with suggestions for future work.

2. Preliminaries

2.1 Petri nets

Petri nets are a formal method with strong mathematical foundation and a graphical representation. The mathematical foundation promotes the use of automatic analysis and verification techniques. On the other hand the graphical notation avoids the use of possibly ambiguous textual notations or hard to understand mathematical notations. Petri nets can be used in the design of complex systems, expressing properties such as precedence relationships, conflicts, concurrency, synchronization, deadlocks, and resource sharing among others. Also, the state and action locality characteristic allow the modeling of complex systems using either bottom-up or top-down approaches. Therefore, it promotes modularity and reusability that are important characteristics for the modeling solution presented in this work.

As mentioned in the introduction, in the context of this chapter an extension of Petri nets called Hierarchical Coloured Petri Nets (HCPN) is used as a description language. This extension incorporates complex data types and hierarchy concepts to Petri nets. An HCPN is a set of non-hierarchical CPN models, and each CPN model is called a CPN page. Therefore, an HCPN is an extension of the concept of CPN that allows the modeling in hierarchical levels. This is possible due to the inclusion of two mechanisms: substitution transition and fusion places. A substitution transition is a transition that represents a CPN page. The page in which the substitution transition belongs to is called *superpage* and the page represented by that transition is called *subpage.* The association between subpages and superpages is done by means of sockets and ports. Sockets are all the input and output places of the substitution transition in the superpage. Ports are the places in the subpage associated to the sockets. The ports can be of input, output, or input-output type. For simulation and state space calculation, sockets and ports are glued together and the resulting model is a flat CPN model.

The fusion places are physically different but logically only one, defined by means of a fusion set. Therefore, all the places belonging to a fusion set have always the same marking.

A marking of a place is the set of tokens in that place in a given moment. And the marking of a net is the set of markings of all places in the net, in a given moment. When a marking of a place belonging to a fusion set changes, the marking of all places belonging to that set also changes.

Indeed, these two additional mechanisms, substitution transition and fusion places, are only a graphical notation that promotes the organization and visualization of a CPN model more efficiently. They favor the modeling of larger and more complex systems because the designer can obtain a model by either abstraction or composition, or even both. In order to manipulate tokens in a CPN, it is defined the concept of multi-set, that is, a set where it is possible to have several occurrences of the same element. This concept allows similar parts of the model to be modeled as token information instead of structure replication.

In the following, the definition of CPN according to (Jensen, 1992) is presented.

Definition 1: A Coloured Petri net is a tuple CPN = $(\Sigma, P, T, A, N, C, G, E, I)$ satisfying the requirements below:

1. Σ is a finite set of non-empty types, called colour sets.

2. P is a finite set of places,
3. T is a finite set of transitions,
4. A is a finite set of arcs such that:

$$P \cap T = P \cap A = T \cap A = 0$$

5. N is a node function defined from A into (P x T) U (T x P).
6. C is a colour function defined from P into S.
7. G is a guard function defined from T into expressions such that:

$$\forall t \in T: [\mathrm{Type}(G(t)) = B \wedge \mathrm{Type}(\mathrm{Var}(G(t))) \subseteq \Sigma]$$

8. E is an arc expression function defined from A into expressions such that:

$$\forall a \in A: [\mathrm{Type}(E(a)) = C(p(a))_{MS} \wedge \mathrm{Type}(\mathrm{Var}(E(a))) \subseteq \Sigma]$$

9. I is an initialization function defined from P into closed expressions such that:

$$\forall p \in P: [\mathrm{Type}(I(p)) = C(p)_{MS}]$$

The definition of HCPN according to (Jensen, 1992) is presented as follows.
Definition 2: A Hierarchical CPN is a tuple HCPN = (S, SN, SA, PN, PT, PA, FS, FT, PP) satisfying the following requirements:

1. S is a finite set of pages such that:
   - Each page s ∈ S is a CPN:
     $(\Sigma_S, P_S, T_s, A_s, N_s, C_s, G_s, E_s, I_s)$
   - The set of net elements are pair wise disjoint:
     $\forall s_1, s_2 \in S: [s_1 \neq s_2 \rightarrow (P_{s_1} \cup T_{s_1} \cup A_{s_1}) \cap (P_{s_2} \cup T_{s_2} \cup A_{s_2}) = \emptyset$
2. SN ⊆ T is a set of substitution nodes,
3. A is a page assignment function defined from SN into S such that:
   - No page is subpage of itself:
     $\{s_0 s_1 \cdots s_n \in S^* | n \in N_+ \wedge s_0 = s_n \wedge \forall k \in 1..n: s_k \in SA(SN_{s_{k-1}})\} = \emptyset$
4. PN ⊆ P is a set of port nodes.
5. PT is a port type function defined from PN into {in, out, i/o, general},
6. PA is a port assignment function defined from SN into binary relations such that:
   - Socket nodes are related to port nodes:
     $\forall t \in SN: PA(t) \subseteq X(t) \times PN_{SA(t)}$
   - Socket nodes are of the correct type:
     $\forall t \in SN, \forall (p_1, p_2) \in PA(t): [PT(p_2) \neq \mathrm{general} \Rightarrow ST(p_1, t) = PT(p_2)]$
   - Related nodes have identical colour sets and equivalent initialization expressions:
     $\forall t \in SN, \forall (p_1, p_2) \in PA(t): [C(p_1) = C(p_2) \wedge I(p_1) <>= I(p_2) <>]$
7. FS ⊆ P_s is a finite set of fusion sets such that:
   - Members of a fusion set have identical colour sets and equivalent initialization expressions:
     $\forall fs \in FS, \forall p_1, p_2 \in fs: [C(p_1) = C(p_2) \wedge I(p_1) <>= I(p_2) <>]$
8. FT is a fusion type function defined from fusion sets into {global, page, instance} such that:
   - page and instance fusion sets belong to a single page:

$$\forall fs \in FS[FT(fs) \neq global \Rightarrow \exists s \in S: fs \subseteq Ps]$$

9. $PP \in S_{ms}$ is a multi-set of prime pages.

Based on Definitions 1 and 2 several activities are defined to manipulate nets using the Design/CPN tool in order to develop and maintain a reuse-based modeling process for complex software system, as detailed in the following sections. Detailed explanations for these definitions, as well as the dynamic behavior of CPN and HCPN are omitted. For more information the reader can see the references (Jensen, 1992. Jensen, 1997).

2.2 Temporal logic

Temporal logic is a modal logic that can be used to describe how events occur over the time. There are operators to describe safety, liveness and precedence properties, providing a framework to specify software systems, particularly concurrent systems (Pnueli, 1977). Temporal logics are used to predicate over the behavior of a system defined by a Kripke structure. This behavior is obtained starting from an initial state and then repeatedly moving from one state to another following the transition relation. It means that such relation should be total, and as consequence all the behaviors of the system are infinite. Since a state can have more than one successor, the structure can be thought of as unwinding into an infinite tree, representing all the possible executions of the system starting from its initial states.

Two useful temporal logics are Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). They differ in how they handle branching in the underlying computation tree. The CTL operators permit to quantify over the paths departing from a given state. In LTL, operators are intended to describe properties of all possible computation paths. It is an agreement that the temporal logic provides a good framework to describe and to reason about the behavior of concurrent systems. However, it is not the case when the question is which one is more appropriate, linear or branching time logic, to do it. But this is a question that is outside of the scope of this chapter. Along this chapter, we use a Computation Tree Logic (CTL) (Clarke et al., 1999) defined for Coloured Petri Nets named ASK-CTL (Christensen and Haagh, 1996). In what follows the basic concepts of both logics are introduced.

The CTL temporal logic combines path quantifiers with linear time temporal logic operators. The path quantifiers A *("for all paths")* and E *("for some paths")* should be used as a prefix of one of the operators G *("globally")*, F *("sometimes")*, X *("nexttime")* and U *("until")*. Let *AP* be set of atomic propositions, then the syntax of CTL is given by the following rules:

1) If $\varphi \in AP$, then φ is a formula, where AP is a set of atomic propositions;
2) If φ_1 and φ_2 are formulae, then $\neg\varphi_1$, $(\varphi_1 \vee \varphi_2)$ and $(\varphi_1 \wedge \varphi_2)$ are also formulae;
3) If φ_1 and φ_2 are formulae, then $EX\varphi_1$, $EG\varphi_1$ and $E[\varphi_1 U \varphi_2]$ are also formulae.

The others CTL operators are expressed using the three operators EX, EG and E[U]. Therefore:

$$AX\varphi \equiv \neg EX\neg\varphi$$

$$EF\varphi \equiv E[\text{true } U\, \varphi]$$

$$AG\varphi \equiv \neg EF\neg\varphi$$

$$AF\varphi \equiv \neg EG\neg\varphi$$

$$A[\varphi_1 U \varphi_2] \equiv \neg E[\neg\varphi_2\ U\ (\neg\varphi_1 \wedge \neg\varphi_2)] \wedge \neg EG\neg\varphi_2$$

The semantic of CTL is defined with respect to paths in a Kripke structure. A path is an infinite sequence of states (*s_0, s_1,...)* such that s_{i+1} is reached from s_i for all $i \geq 0$. So, if φ is a CTL formula $M, s \vDash \phi$ is used to denote that φ holds for s_0 of M.

The four most used CTL operators are *EF, AF, EG*, and *AG*. In Fig. 1 the interpretation for such operators is illustrated in an intuitive way, and they are interpreted as follows.

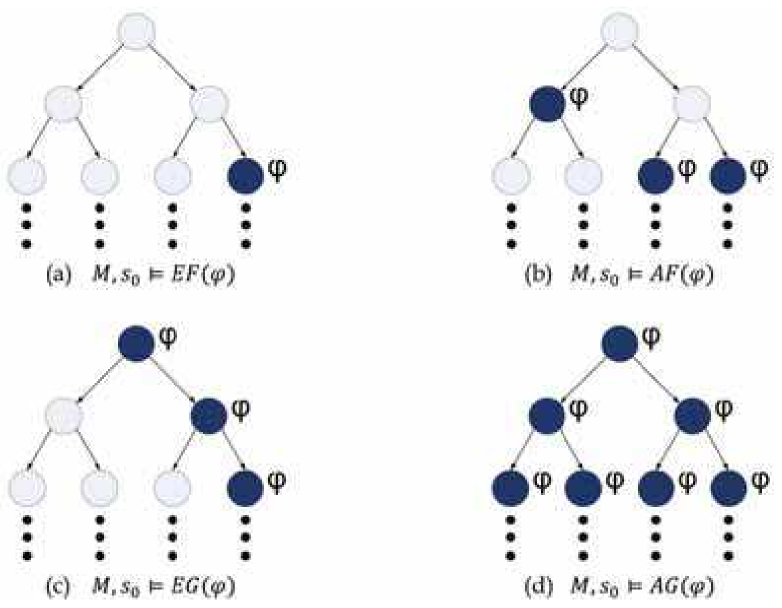

φ
(a) $M, s_0 \vDash EF(\varphi)$
φ
φ
φ
(b) $M, s_0 \vDash AF(\varphi)$
φ
φ
φ
(c) $M, s_0 \vDash EG(\varphi)$
φ
φ φ
φ φ φ φ
(d) $M, s_0 \vDash AG(\varphi)$

Fig. 1. Basic CTL operators.

EFφ ≡ E[true U φ] means that exists a path starting from s_0 in which φ holds at some state along this path.

AFφ ≡ A[true U φ] means that for all paths starting from s_0, φ holds at some state along the path. In other words, φ is inevitable.

EGφ ≡ ¬AF¬φ means that exists a path starting from s_0 in which φ holds at every state along this path.

AGφ ≡ ¬EF¬φ means that for all paths starting from s_0, φ holds at every state along that paths. In other words, φ holds globally.

2.3 ASK-CTL

ASK-CTL is a CTL-like logic useful to specify properties for CPNs (Coloured Petri Nets) state spaces, represented by occurrence graphs. Occurrence graphs carry information on both nodes and edges. Hence, a natural extension for CTL is to include the possibility to

express properties about the information labeling for the edges (e.g., edge information is needed when expressing liveness properties since liveness is expressed by means of transition occurrence information). For this purpose two mutually recursively defined syntactic categories of formulae are defined: state and transition formulae, which are interpreted over the state space for states and transitions respectively (Cheng et al., 1997). As in CTL, quantified state formulae and transition formulae are interpreted over paths. Path quantification is used in combination with the *until* operator to express temporal properties.
The ASK-CTL library has two parts: one which implements the ASK-CTL logic language and another one which implements the model checker (Christensen and Haagh, 1996). The syntax of ASK-CTL is minimal and in order to increase the readability of the formulae we make use of syntactic sugar, e.g., POS(φ) means that it is possible to reach a state where φ holds, INV(φ) means that φ holds in every reachable state, and EV(φ) means that for all paths φ holds within a finite number of steps.

2.4 Model Checking

The need to increase the dependability of software systems motivates the definition and application of more dependable developing methods and techniques. This need is more evident when dealing with critical real-time systems. With the increasing complexity of the systems the traditional methods based on tests, for example, are not enough anymore to guarantee dependability.
The use of formal methods can increase the confidence in the behavior of the system. In the specification, formal methods can be used to find difficult errors before developing the real system. Traditional methods based on tests and simulation can detect initial errors. But after the simplest errors are fixed, more rigorous methods are needed.
Model checking is used to verify specifications (Clarke et al., 1999) in an exhaustive way. That is, where tests and simulations analyze some possibilities, formal methods analyze all possible behaviors.
One great advantage of model checking is that it is fully automatic. Moreover, the model checking algorithms generate a counter-example generation in case of negation of a property indicating a path where the property is false.
The disadvantage of model checking is the state explosion problem. That occurs when the system has several concurrent components, or when it manipulates complex data types. Some techniques have been researched and developed to deal with this problem such as symbolic model checking (McMillan, 1993) and partial order reduction (Peled, 1994. Valmari, 1991).
The verification activity consists in checking if a property is satisfied by a model or not. The properties are described in temporal logic, and the models can be described as a finite automaton or as a Petri net, for instance. Let M be a model and f be a temporal logic formula that express some property of M. The model checking consists in verify if M models f, which is noted by $M \models f$.
The model checking consists of the following three activities:
Modeling: The modeling consists in describing the system in some formalism. The formalism to be used depends on the tool to be used in the verification, the designer knowledge, or the culture in the institution that is developing the project. It is still possible to transform a given formalism into another to perform the verification.
Specification: The specification is usually done in temporal logic that is used to specify how a system's behavior evolves over time. It is not possible to guarantee the completeness of the specification, that is, it is not possible to guarantee that all the properties to be verified

are specified. But once a property is specified it can be checked against the model for all its possible behaviors.
Verification: Given a model and a specification the verification is fully automatic. In the case of a property is negated the designer must analyze the counter-example to solve possible modeling errors, or to reformulate the specification. Moreover, abstraction and modular techniques depend on the designer to allow that the verification can be performed dealing with the state explosion problem.

3. Reuse based software modeling

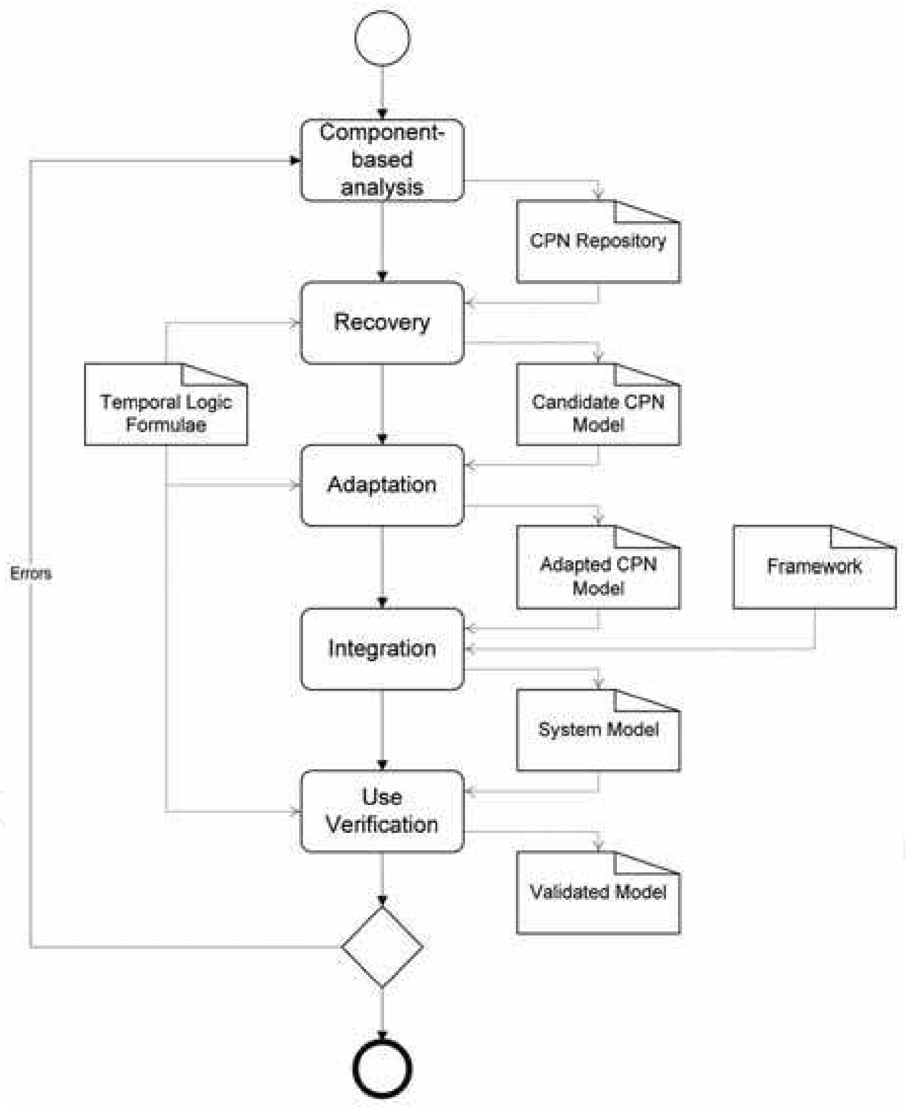


Fig. 2. Diagram for the systematic reuse solution.

When using a reuse based modeling method it is not always necessary to build the hole system from the scratch, it is possible that some of the required models of the system are already modeled. The reuse process defined in this chapter is illustrated by Fig. 2. The main reuse activities are recovery, adaptation, integration and use verification. A detailed discussion of the repository management activity, recovery and insertion of models, can be found in (Lemos & Perkusich, 2001). In this work the adaptation of a recovered model (Gorgdnio & Perkusich, 2002) and the integration of such model into an architectural framework are detailed. The functionality of the reuse process as a whole, unifying all the activities in a systematic modeling method is also discussed.
Besides the reuse activities, a use verification step is taken into account. This step consists in performing model checking in the integrated models in order to verify whether the specific use case is correct, that is, to verify if they were correctly used.
As pointed out in the introduction, an embedded system is used to illustrate the process as well as to guide the definition of the activities.
The designer must think on how and where to search for models that can be directly reused, and adapted if necessary, while building a new system. Moreover, the designer must try to identify potential candidates for reuse and store them in a repository of models. The following reuse activities are identified during the formal modeling of systems:

- Identification of the parts of the new model;
- Definition of a framework;
- Detection of the parts that need to be constructed and those that can be reused;
- Description and recovery of the models that can be reused;
- Adaptation of the recovered models;
- Integration of the recovered/adapted/constructed models;
- Identification of new reusable models and storing them in the repository

It is important to point out that, besides the fact that this technique is fully supported by a set of tools the methodology itself is not completely automatic. The designer plays an important role and is required to create the framework on which the recovered and adapted models are integrated. Moreover, she is required to write down a set of temporal logic formulae describing the behavior of the models to be recovered and adapted.

4. Case study: A component-based embedded system

The application domain considered in the scope of this work is an embedded transducer network control system (Silva & Perkusich, 2005). As shown in Fig. 3, this system is composed by a set of transducers, a controller, named the communication server, and a real-time server.
The environment signals acquired by the sensors are transformed and controlled in a way that the real-time server can access and modify the information to control the actuators according to the application requirements. Observe that the transducers are connected to a controller and that besides control functions it also acts as a front end communication server. Therefore, different applications can be specified and verified by changing the components. Several different applications may access the real-time server to acquire data and to control devices. For instance, it is possible to have temperature, ventilation and humidity sensors. The signals that are acquired and processed can be used to control an HVAC (Heating, Ventilation and Air Conditioning) system in an intelligent building.

It is important to note that a system defined as shown in Fig. 2 is very common in many other kinds of command and control systems and therefore it is possible to define a software architecture that can be reused in other applications.

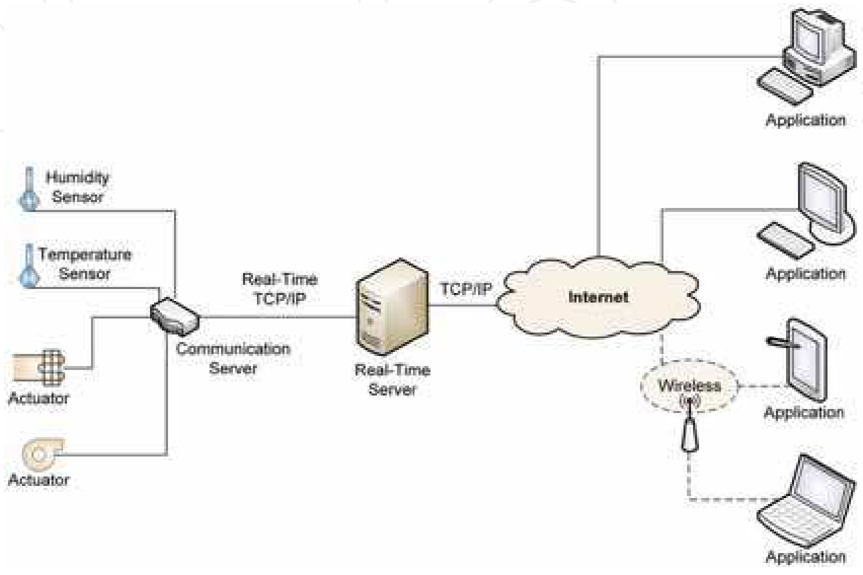


Fig. 3. System structure.

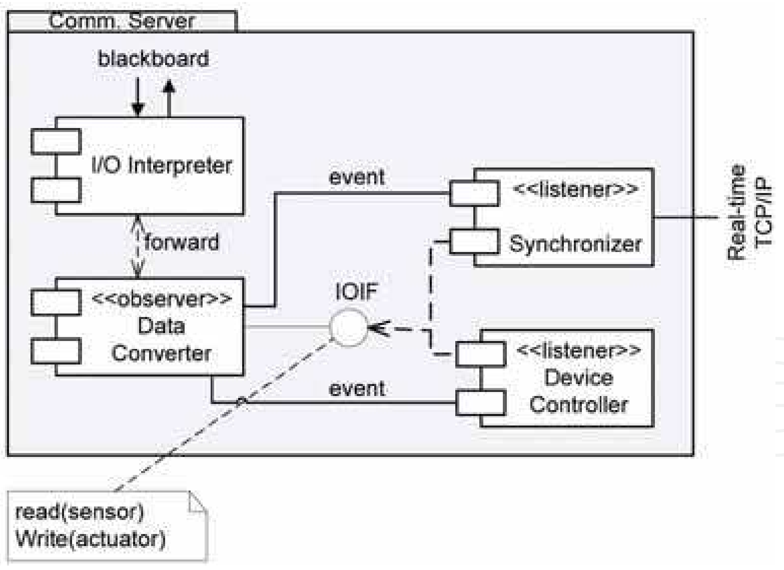


Fig. 4. Communication server

According to the requirements of the applications, defined based on the command and control problem, and the transducers used, different systems can be built. Based on the

approach introduced in this work a system does not need to be specified and verified always from the scratch. What is necessary to do is to recover a model from a repository, modify it to satisfy the new requirements and integrate it on the new project. Only in the case that no model can be found, the designer must specify a new one. To promote this approach, a product line to this specific domain using some specific software architecture to reuse common components is defined. The main advantages are time and money savings, and the reduction or even elimination of errors, and therefore, faults can be avoided. Also, it is possible to maintain and evolve a repository of reusable components for a given domain improving the dependability on the models.
An important observation s that the details related to specific technologies to implement the components are abstracted. The focus is on the specification and analysis of the architecture of a target system. Therefore, properties for the interfaces and architectural level of the components are verified regardless internal details of them. For instance, the protocol used by the communication server to communicate with the control system running in the real-time server is abstracted.
In Figs. 4 and *5 is* illustrated the component diagrams for the communication server and the real-time server, respectively. The communication server consists of four components. The IO Interpreter instantiates raw data from sensors to objects. The Data Converter transforms the data to an specific format. Device Controller is used to calibration, initialization and other control tasks. The Synchronizer is the communication channel with the real-time server. The real-time server is composed by three components. The data controller is used to control data flow among several applications accessing the server. The access to the server is available through the UI component. The real-time server Synchronizer is the counterpart to the communication server Synchronizer.

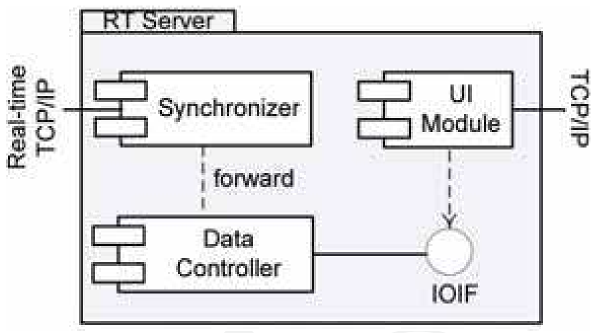


Fig. 5. Real-time server

4.1 Framework

In Fig. 6 the HCPN framework that specifies the architecture of the system is illustrated. There, it is possible to identify how the entities communicate with each other. The System page models the sensors and actuators. They communicate only with the communication server represented by the CommServer page.
There are several components defined for the communication server page. Data from the devices to the embedded system and output data to devices are communicated using a blackboard mechanism. The input and output interpreter, lOlnterpreter, is used to instantiate the data written in the blackboard as objects. Also, this component receives objects from the system and translates them to the data format used by the devices. This component is fixed in the architecture. That is, it is not necessary to change it from one project to another.

The next component is the data converter, DataConvert. This component transforms data from the I/O interpreter to a format used by the real-time server, according to the requirements of the applications. Since data formats are dependent of the applications that access the server this component must be changed to satisfy the requirements of each project. The data converter decides the data flow. If data in the data converter is a control requisition, such as an initialization or calibration request, that data is sent to the device controller. If data is an information signal it is sent to the synchronizer, EmbChannel, to be transmitted to the realtime server.
The device controller component, DeviceControl, is used to control devices, that is, as said before to perform calibration and initialization tasks. Moreover an application can request changes in the attributes for a device, such as, the sampling time. This is done also by the device controller. The synchronizer is a realization of the communication between the communication server and the real-time server. When a sensor sends an information signal and not a control signal, it must be transmitted to the real-time server through the synchronizer. Thus, there is a synchronizer for the communication server and another one for the real-time server. Since this communication does not change, the synchronizer is fixed in the architecture.

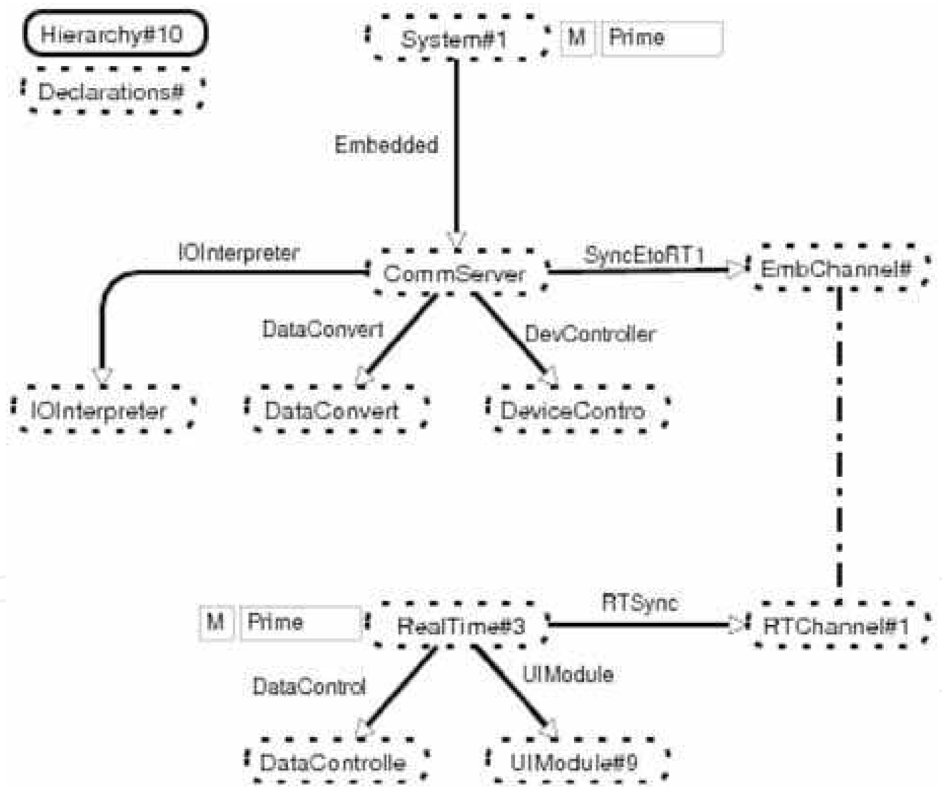


Fig. 6. Model hierarchy (framework).

The real-time server, RealTime, intermediates the communication between the communication server and the applications. A database with information about the net and the applications is used to promote this communication. The applications can read or write information to control the system. In the real-time server we have several components also. The synchronizer, RTCHannel, is identical to the one in the embedded system. The data controller, DataContoller, is used to control data flow from and to the applications. The user interface module component, UIModule, is used to make services available. The applications use this component to access the system.

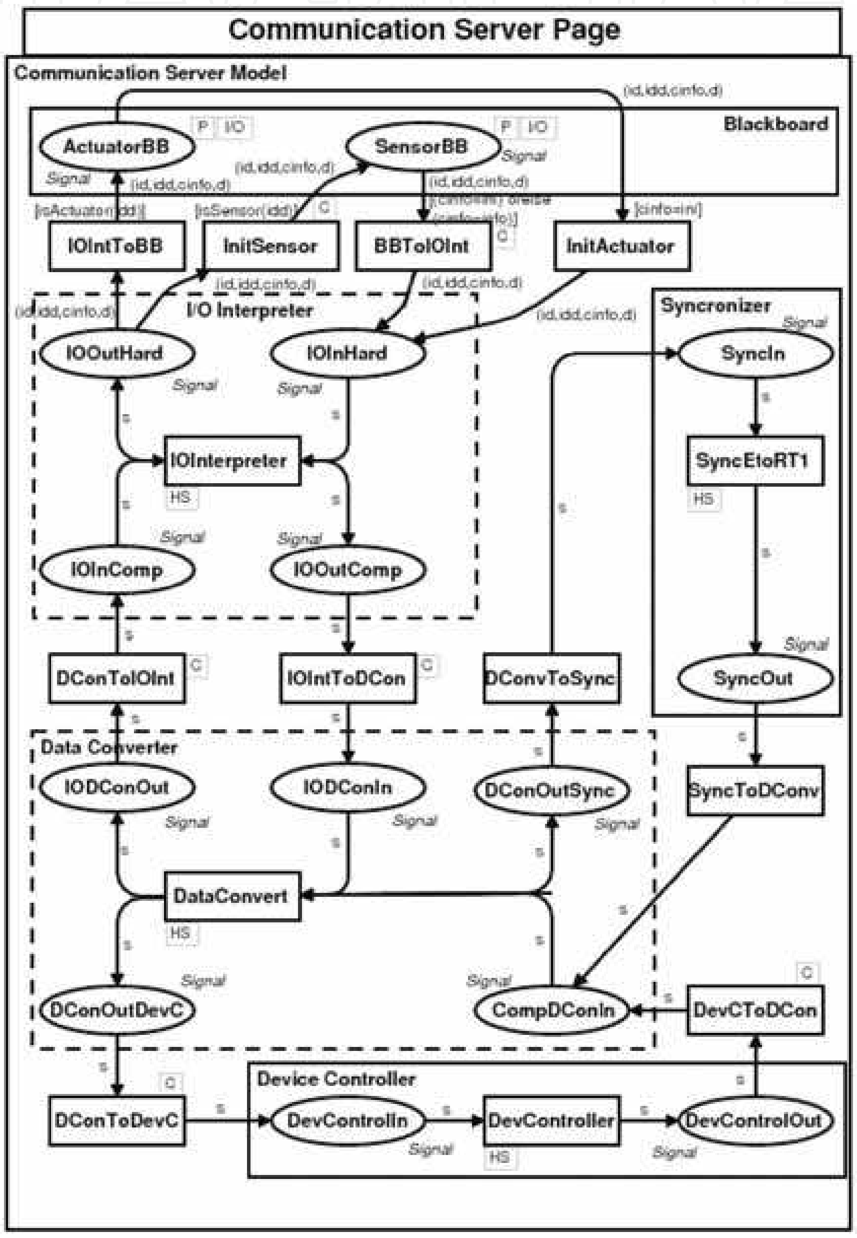


Fig. 7. Embedded system.

In Fig. 7 the dashed lines define components that must be replaced, or hot spots, based on the application, and the continuous lines define components that do not need to be changed or frozen spots. It is possible to see all the components for the communication server. The top part is the blackboard where messages are exchanged between the sensors and actuators and the server. The first component is the I/O interpreter. After this component the signal is sent to the Data Converter and at this point it can take two different destinations, the Device Controller, or the Synchronizer.
Using this architecture, it is possible to specify any control system as defined here, promoting a product line evolution based on the reuse of component models. Moreover, this strategy allows the practice of refactoring at a model level.
The specification described in this section is a general explanation of the model. This model was constructed using the reuse process described in Section 3.

5. Model adaptation

Once a model, that is a candidate to be reused, is identified and recovered from the repository, it is necessary to verify if it is ready to be integrated on the system framework. Usually it is necessary to adapt the recovered model to satisfy some special conditions that holds in the new system. The adaptation technique presented here is based on the use of temporal logic, model checking and supervisory control theory. The basic idea is that for a given CPN model that satisfies some properties, it should be possible to refine it in order to obtain a new model whose behavior is a refinement of the behavior of the original one. Note that on the context of this work, adaptation is a refinement relation. Basically, it means that all the possible behaviors of the adapted model are also allowed in the original model, and in some sense, the models can be related trough a preorder relation (Long, 1993).
In (Ramadge and Wonham, 1987) an algorithm to obtain the supremal controllable sub-language for a given language is described. They assumed that a system, described as a finite automaton, is composed of some events that can be controlled and others that cannot. If the occurrence of a controllable event leads to an undesired situation, it is possible to disable it. However, if the event is not controllable, then it is not possible to do it. For example, in the case of the environmental controller the changes in the temperature of the room, i.e. the data received by the sensors, are not an event that can be controlled by the system. And it makes no sense to change the behavior of the model by avoiding the occurrence of an event that cannot be controlled.
The supremal controllable sub-language algorithm receives as input two finite automata. One modeling the system, m_1 and other modeling the desired behavior of the system, m_2, and the set of events is divided into controllable and uncontrollable. The algorithm returns an automata m_3 that is the maximal, with respect to the behavior allowed by m_2, automata that can be controlled without reaching any undesirable state. In general terms, the algorithm works by removing the undesired states from m_1 until a fix point is reached.
The problem with this direct approach is that it is not possible to know in advance if such m_3 exists or not. It may be the case that it is not possible to refine m_1 until the resulting model satisfies the properties specified by m_2. This is known only after the execution of the algorithm when it returns an empty model. Since it is executed over the state space of the models, if such state space is too large, it is possible that the entire process takes a long time to be executed without generating any useful output.

Model checking techniques can be used to avoid this problem. The idea is to model the automata m_2 as set of temporal logic formulae, more specifically CTL, and use them to check if m_1 can be refined to satisfy the CTL properties or not. Note that it is necessary to determine if there is a subset of m_1 that satisfies the properties. If the model checker outputs a positive result, the synthesis procedure is executed.
Observe that the models to be adapted are given as CPN models. So, the first step in the adaptation procedure is the generation of its state space. Then, it is possible to verify and refine it as described above.
After the execution of the supremal controllable sub-language algorithm, an occurrence graph that is isomorphic to the original one is obtained. The only difference is that the states that should not be reached are marked as undesirables. They are not removed from the occurrence graph of the Petri net.
Next it is necessary to modify the input CPN model in such a way that the state graph of the new model will be isomorphic, considering the label of each arc, to a sub-graph of the original state graph of the input model. Once it is done, the new CPN model is generated.

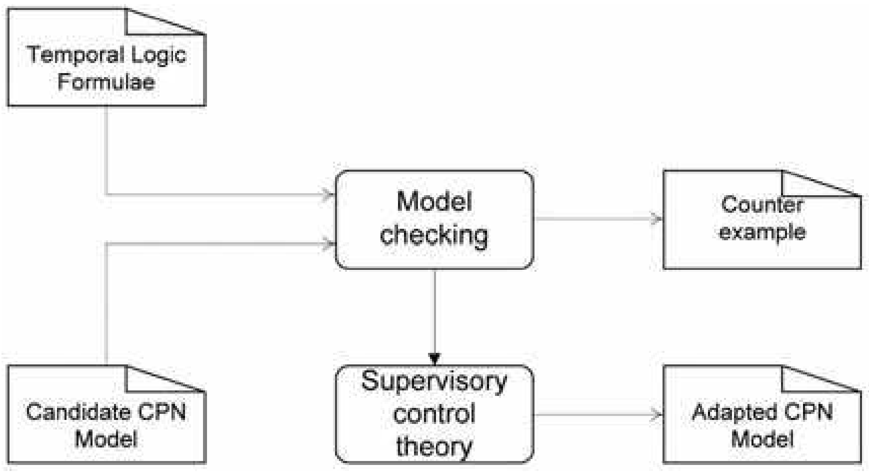


Fig. 8. Adaptation of reusable models.

The adaptation technique introduced here is illustrated in Fig. 8 and the steps required to perform the adaptation are:

1. Generate the occurrence graph of the CPN model;
2. Verify if the CPN model may satisfy the new specification by applying model checking techniques;
    - In the negative case, ask for human intervention and terminate;
3. Execute the supremal controllable sub-language algorithm;
4. Adapt CPN model to generate the new occurrence graph.

Adapting the CPN model consists in adding some control information on it in such a way that the states marked as undesirables are not reached in the new model. In the Design/CPN, each state of the occurrence graph of a model has a unique label. Taking this into consideration, control can be added to the CPN model by creating a new place, called control place, which should be input and output place of all transitions of the CPN model. The initial marking of this control place will be the label of the initial state of the adapted

occurrence graph. Every time a transition occurs, it removes the token on the control place and puts a new one with the label of the new reached state.
The value of the new state is determined by a function that is constructed from the adapted occurrence graph. This function receives as input the value of the token in the control place that represents a state s in the adapted graph and the label t of the transition being executed. The output is the label of the state s' in the adapted graph that is reached from s through the occurrence of t. This control function should be attached to all transitions in the model, i.e. every time a transition occurs the function is executed and the value of the token in the control place is updated.
Finally, it is necessary to add guards to some transitions of the CPN model in order to disable them if its execution in the current state leads to an undesirable state. Note that a transition may be enabled or disabled depending on the value of the marking of the control place. Therefore, if $s_i \xrightarrow{t} s_j$ belongs to the adapted state graph, s_j is marked as an undesirable state, and t models a controlled event defined for the system, a guard is added to transition t to disable it whenever the marking of the control place is the label of s_i. Observe that if t models an uncontrollable event, no guard can be added to it. However, t should always be connect to the control place due to the fact that even if t cannot be controllable, it should be observable.

5.1 Implementation

Algorithm 1: Model Adaptation.

```
1.  OCCGRAPH := MakeStateSpace(CPN);
2.  SCCGRAPH := MakeSCCGraph(OCCGRAPH);
3.  if ASKCTL(properties) == true then
4.      NEW_OCCGRAPH := RamadgeWonham(OCCGRAPH);
5.      CP := CreateControlPlace(CPN);
6.      SetInitialmarking(CP);
7.      CF := CreateControlFunction(NEW_OCCGRAPH);
8.      AddToGolbalDeclarationNode(CF);
9.      for all Transition $t$ in CPN do
10.         AddNewArc(CP, $t$)
11.         AddNewArc($t$, CP);
12.         AddCodeSegmentToTransition($t$, CF);
13.     end for
14.     for all State $M$ in NEW_OCCGRAPH not marked as undesirablr do
15.         for all $M'$ such that $M \xrightarrow{t} M'$ do
16.             if $M'$ is marked as undesirable and $t$ is controllable then
17.                 AddGuardToTransition($t$, label($M$));
18.             end if
19.         end for
20.     end for
21. end if
```

Before presenting the algorithm, the adaptation problem can be stated as follows:
"*Given a CPN model, called CPN, and a set of behavioral restrictions described as CTL formulae, the adaptation problem consists in the synthesis of a new CPN model, called CPN_{adp}, that satisfies these new restrictions taking CPN as the starting point.*"

The adaptation strategy described above is defined by Algorithm 1, which is implemented in CPN/ML (Christensen and Haagh, 1996. Ullman, 1998), and it is executed inside the Design/CPN tool as a loadable module.
Lines 1 and 2 define the computation of the occurrence graph (OCCGRAPH) and the strongly connected components graph (SCCGRAPH) of the CPN model. Lines 3 and 4 define the invocations of the ASKCTL model checker and to the implementation of the Ramadge and Wonham algorithm respectively. The other functions are defined to manipulate the internal structures of the Design/CPN models to add the objects used to add control to the model. In Fig. *9* it is illustrated how the adaptation can be performed for the specification of a system.

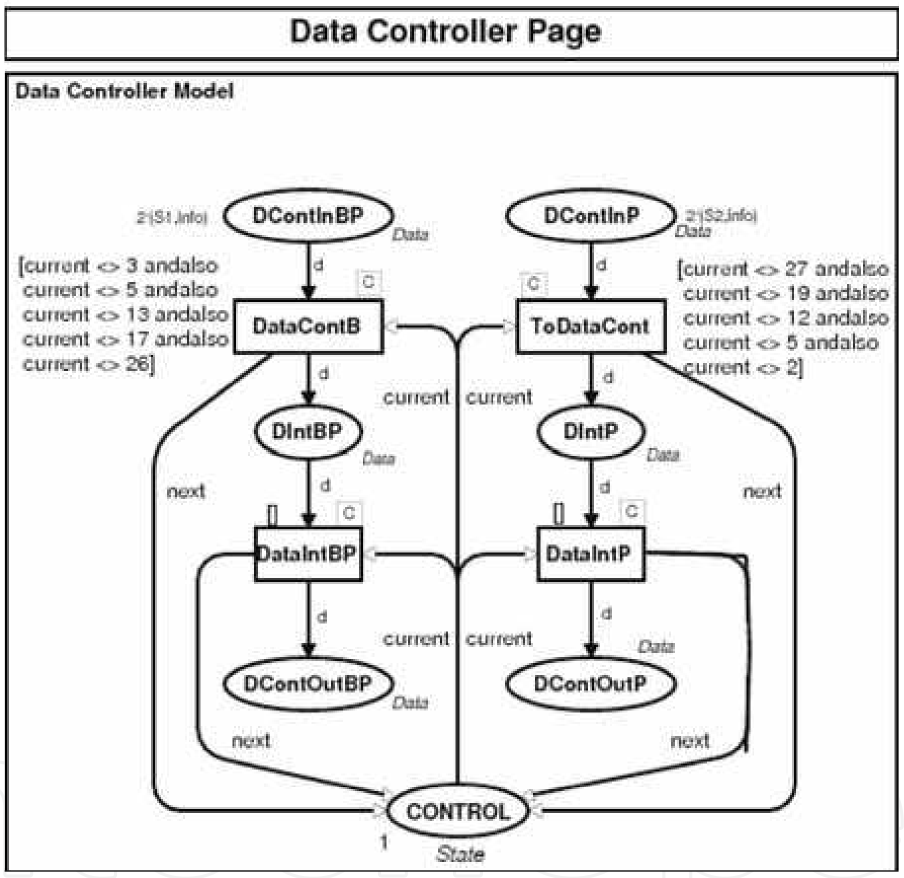


Fig. *9.* Adapted CPN model.

For this example, the component DataController (see framework on Fig. 2) is adapted in order to limit the capacity of the output buffers, DIntBP and DlntP, in such a way that each of them never store more than one token at a given time. This property is expressed in the ASKCTL language by formula:

POS(NOT(MoreThanOne));

Such formula captures the notion that there is a path in the OCC graph for which the evaluation of the function MoreThanOne is false. Such function receives as input a node of the OCC graph and checks if the number of tokens in the places DIntBP and DlntP are greater than one. MoreThanOne is also written in CPN/ML, and it makes explicit references to the elements of the component DataController. This means the designer does not see the component as a black box, and she should have some acknowledgment of the model candidate to be reused.
The code bellow is part of the control function generated by the adaptation procedure. This function takes the label of the current state of the CPN model as input and returns the label of the next state. This information is used to decide whether a transition can be allowed to fire or not.

```
var current: STATE;
var next: STATE;
fun NextState(trans, mark)=
    case (trans, mark) of
        ("DataController'DataContB",   1) => 3 |
        ("DataController'ToDataCont", 1) => 2 |
        ("DataController'DataContB",   3) => 6 |
        ("DataController'ToDataCont", 3) => 5 |
        ("DataController'DataContB",   5) => 8 |
        ("DataController'ToDataCont", 5) => 7 |
        ("DataController'DataContB",   2) => 5 |
        ("DataController'ToDataCont", 2) => 4 |
        ...
```

And finally the guards of transitions DataContB and ToDataCont are added in order to disable them when necessary. Observe that there are no guards on transitions DContOutBP and DContOutP. This is due to the fact that there no situation in which their occurrence leads to an undesirable state, i.e. they cannot increase the amount of tokens in places DlntBP and DlntP.

6. Model integration

After a model is recovered from the repository, and possibly adapted, it has to be integrated into the framework. The integration, as well as the other activities, are fully implemented using the CPN/ML language (Christensen and Haagh, 1996) for the Design/CPN tool set.
First, the designer is asked the name of the file with the CPN model to be integrated into the framework. Then, some functions are automatically executed to build the integration environment, that is, the places, transitions, arcs, and its respective names, color sets, and inscriptions. The next step executed by the algorithm is the definition of the input and output ports in the diagram being integrated. After this step, the substitution transition is defined, and the sockets in the superpage are associated to its respective ports, previously defined in the subpage. The last step is to select the box with the model declarations, in the model page, to define the color set of the ports based on the sockets colors, and to append this information in the global declaration node.
The file selection needs the user interaction, while all other steps are fully automatic executed. To define the algorithm some restrictions are considered. They are:

- Unique page name;
- Prefix in the color sets names indicates the page name;
- Suffix in the place names indicates whether it is a port or not;
- Dot-dashed line patter for the auxiliary box with model declarations;
- Model declarations box must to be unique;
- Declarations of the port places must be the first ones in the model declarations box.

The first restriction to be considered ensures that the page name of the integrated model is unique, and it is the responsibility of the designer. The model being integrated must have a prefix in its color set names with the page name. Another restriction is about port places. The places that are ports must have a suffix IN or OUT, to input and output ports, respectively, in its names. This restriction is to allow the algorithm to recognize which places are ports and which type of port they are.

The last integration restriction is that in the model being integrated there must exist an auxiliary box with dot-dashed line pattern. This box must contain all the declarations for this page. The declaration of the ports color set must be the first ones in this box. This is necessary to the algorithm be able to adapt correctly the color set to successfully integrate the model.

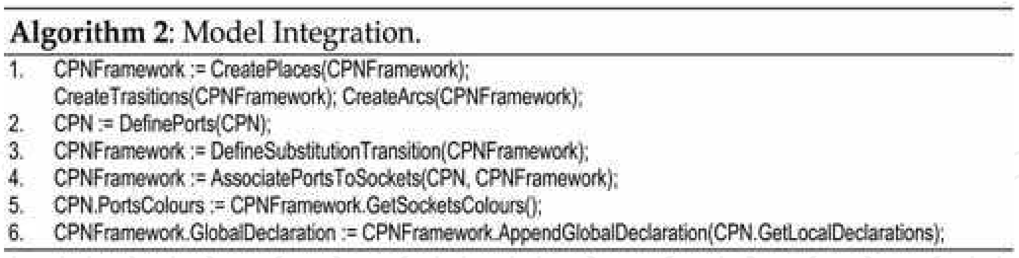
It is important to note that it is the responsibility of the designer to ensure that the restrictions are satisfied in order to the algorithm works properly.

6.1 Implementation

Before presenting the algorithm, the integration problem can be stated as follows:

"*Given a CPN model called CPN, and a Framework called CPNFramework, the integration problem consists in creating places, transitions, arcs, and their respective inscriptions, as well as the hierarchy and append the declarations of CPN to the global declarations of CPNFramework to have a new integrated model taking CPN and CPNFramework as the starting point.*"

The integration is implemented as defined by the Algorithm 2. Steps 2 to 7 are fully automatic. Step 1 needs the user interaction to select the file name of the model to be integrated.



Algorithm 2: Model Integration.

```
1.  CPNFramework := CreatePlaces(CPNFramework);
    CreateTrasitions(CPNFramework); CreateArcs(CPNFramework);
2.  CPN := DefinePorts(CPN);
3.  CPNFramework := DefineSubstitutionTransition(CPNFramework);
4.  CPNFramework := AssociatePortsToSockets(CPN, CPNFramework);
5.  CPN.PortsColours := CPNFramework.GetSocketsColours();
6.  CPNFramework.GlobalDeclaration := CPNFramework.AppendGlobalDeclaration(CPN.GetLocalDeclarations);
```

The integration depends on the framework. Therefore, for each application domain it is necessary to define the architecture of the system and to model it as a CPN framework. For each domain and framework, it is necessary to implement specific functions for the integration step of the reuse process. But this implementation has to be done just one time, and it is used throughout the evolution of the product line in the specific domain with the framework.

Now, it is illustrated how the integration phase of the process is performed. In Fig. 10 it is shown an example where the DataController is integrated into the CommunicationServer model. In this example it is possible to see part of the integration phase, because some parts are internal to the algorithm and to the global declaration node. It is possible to see, for example, how the sockets in the superpage are associated to the ports in the subpage, and how a substitution transition in the superpage represents another CPN model, the page of the component model.

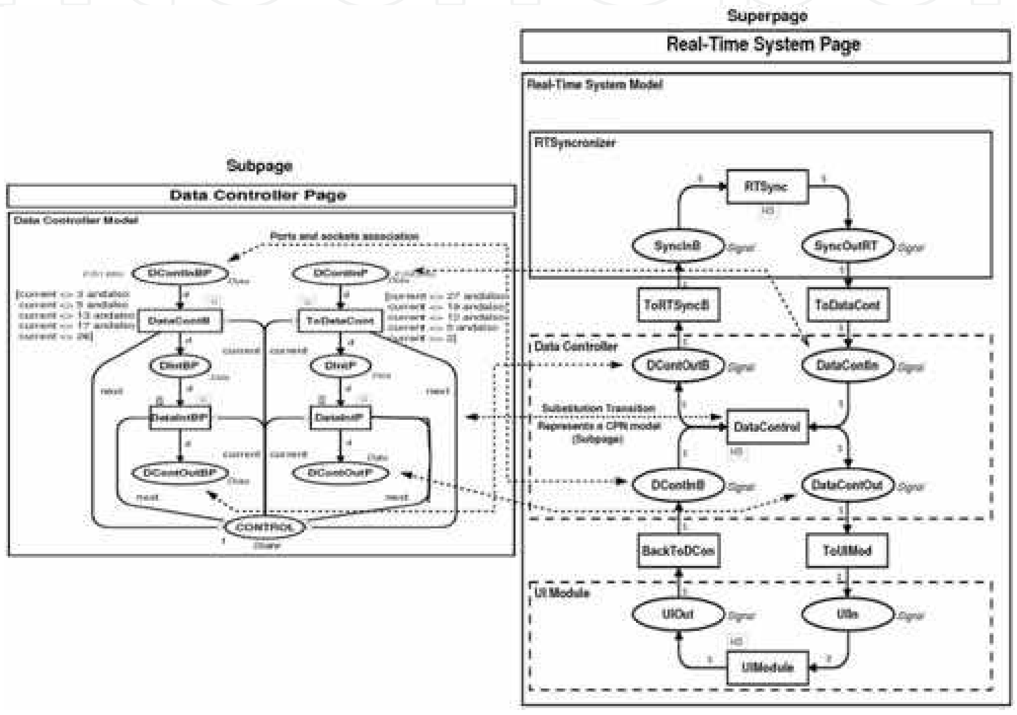


Fig. 10. Integration example.

7. Use verification

Besides adaptation and integration, the specific use case of recovered models performing an use verification step is also considered. This activity is defined in the context of this work because when modeling based on reuse it is necessary to guarantee that the semantic of the resulting model respects the semantics of the reused models. Some parts of the resulting model can lead a reused model to behave in a different way than expected. This problem can compromise the modeling activity and the facility, and flexibility that the reuse process promotes.

The use case verification activity consists in performing model checking for the framework with the individual models to be verified already integrated on it. To do this, the temporal logic formulae for the properties of an individual model is specified in a file,. The model checking is then performed in the whole model to ensure that the integrated models were correctly used. The idea is to use the same specifications used in the recovering step, or the specification supplied together with the model.

Considering that it is necessary to define a new framework for each modeling domain. The framework can be developed without any considerations, or even violations, to the specific functionalities of the individual models to be integrated following the reuse process. Moreover, the models can be reused in several different domains. The interface between the framework and the models can be changed to reflect the needs of each domain and to satisfy the specific use alternatives of each model. Therefore, as the framework is being built the changes in the interface can result in a wrong use case of the integrated models. Because of the problems discussed above, it is necessary to define the use verification activity for the reuse process. Indeed, the use verification activity is essential when building models from a new framework.
Another justification for the definition of the use verification activity is that in the case that no reuse candidate is found in the repository, a new model has to be built. The use verification is also performed to validate a new model to be inserted in the repository.

7.1 Implementation

Before presenting the algorithm, the use verification problem can be stated as follows:
"*Given a CPN model, called CPNFramework, and some properties of an individual component model integrated on CPNFramework, described as temporal logic formulae, the use verification problem consists in performing model checking to verify if the new integrated model respects the individual properties of the integrated component taking CPNFramework and the properties as the starting point.*"
In Algorithm 3 the use verification is defined. Initially, the designer must execute the occurrence graph tool in the Design/CPN. The next step is to select the file that has the properties specifications for the model to be verified. The algorithm executes at this point the model checking in the framework including all the models already integrated on it. If the properties hold in the resulting model the use verification is successful. In the opposed case the designer receives a warning that there exist errors in this specific use case.
The steps 1 to 3 must be done by the user. All other steps are automatically executed. After the execution, a message is shown saying whether the properties are satisfied or not.

Algorithm 3: Use Verification.

```
1.  DELETEOCCGRAPH;
2.  OCCGRAPH := MakeStateSpace(CPNFramework);
3.  SCCGRAPH := MakeSCCGraph(OCCGRAPH);
4.  if ASKCTL(properties) == true then
5.      ShowMessage(SUCCESS);
6.  else
7.      ShowMessage(FAILURE);
8.  end if
```

When the SCC graph is generated and the ASK-CTL library loaded, the use verification is performed. To do this, it is necessary to specify the properties we want to prove. At this moment the designer is asked for the file name with the specifications. After the designer indicates this file name, the model checking is performed.
An important task in verifying systems is to identify and define properties to be proved. As the goal is to prove properties for the architecture and for the interface of the components, the framework model should be used regardless the internal component details to prove properties about the whole system. In order to do that, a technique to identify functional system scenarios is used to define interesting properties to prove. To illustrate the scenarios

Message Sequence Charts (MSC) are used. MSCs are automatically generated during the simulation of the HCPN model. The verification strategy defined in this work consists of the following steps:

1. Identification of scenarios;
2. Automatic MSC generation for each scenario based on simulation;
3. Properties identification based on the MSCs and scenarios;
4. Atomic propositions and formula specifications in temporal logic;
5. Model checking.

When performing the use verification activity of the reuse process described in Section 3, only steps 4 and 5 are done. The strategy above is a more generic approach to verify models of systems, specifically for planning and flow properties. In the rest of this section the verification is considered as the more general approach, but the reader must have in mind that the use verification activity for reused models of specific components are captured by steps 4 and 5.

Suppose that the devices send an initial signal when the system is turned on or a new device is plugged in. Also, suppose that the system perform some task when it receives this kind of message and send back to the device an acknowledgement, calibration, or even an initialization message. Such scenario is illustrated in Fig. 11.

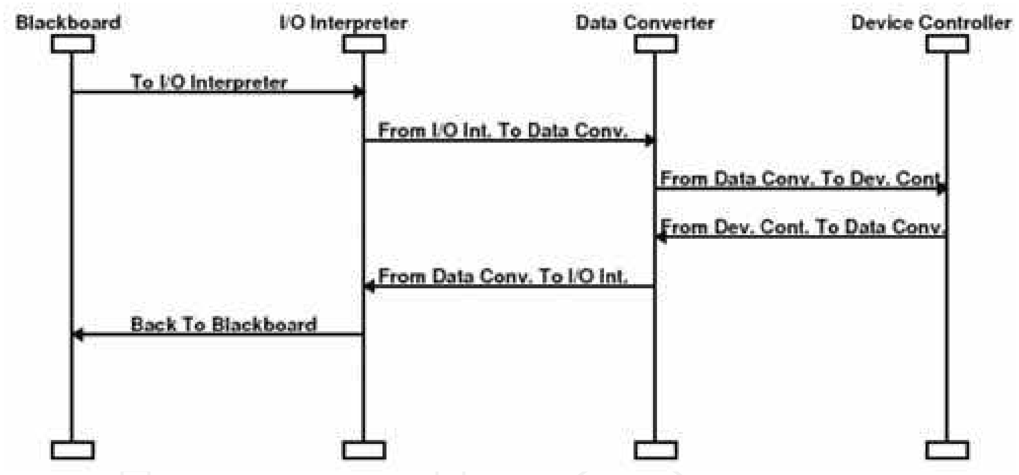
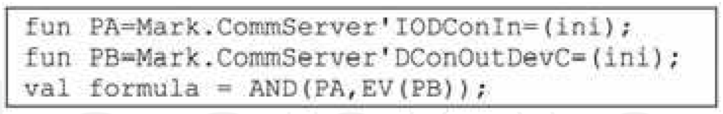

Blackboard
I/O Interpreter
Data Converter
Device Controller
To I/O Interpreter
From I/O Int. To Data Conv.
From Data Conv. To Dev. Cont.
From Dev. Cont. To Data Conv.
From Data Conv. To I/O Int.
Back To Blackboard

Fig. 11. Data converter flow to control signal.

When a device sends an initialization signal, the data converter sends it to the device controller to perform the associated control tasks. Since a MSC diagram is generated based on simulation it captures only a single execution sequence for the information flow in the model. It might be the case that there is some situation where this expected sequence or flow is violated. Therefore, the scenario for all possible model behavior must be verified performing model checking to guarantee that the expected flow is always respected. Considering again the scenario shown in Fig. 10, the property that when an initialization message is sent by some device the flow should be through the device controller must be provedL The following ML code fragment specifies atomic propositions and the temporal formula to prove this property.



```
fun PA=Mark.CommServer'IODConIn=(ini);
fun PB=Mark.CommServer'DConOutDevC=(ini);
val formula = AND(PA,EV(PB));
```

The proposition *PA* is true if there is a token in place lODConln of page CommServer. The proposition *PB* is to true if there is a token in place DConOutDevC of page CommServer. Therefore, the formula is true if PA is true and eventually PB is true. It means that if there is a token in the data converter input, this token is sent to device controller input, which is the component that implements control tasks such as initialization, calibration, and changing devices working parameters. The evaluation of this formula to true means that this part of the model behaves as expected for all possibilities of model execution. We can proceed with the same reasoning to prove that the flow of information back to device also behaves as expected for all possibilities.
The strategy illustrated above can be used for several different properties. It is used together with the framework, and the reuse of CPN models process to promote a formal and systematic approach for the specification and verification of systems.

8. Concluding remarks

In this chapter a component-based software development approach using Coloured Petri Nets (CPN) was presented. Components promote the software development using building blocks as in other engineering disciplines. In order to develop component-based systems it is necessary to define a process.
The implementation of a reuse-based modeling process for CPN, including all the activities in a fully automatic way was introduced. The model reuse activities considered were adaptation, integration and use verification. For different application domains only the integration step is affected because of the need to define different integration strategies and a specific framework. Due to the use of the framework concept, the designer does not need to explicitly care about the model structure.
The application of the introduced methodology for the design of dependable software is natural in a context in which adaptation means refinement. The introduced approach is very useful whenever the CPN model is used in order to provide control information to some other system. The approach introduced in this work has been applied to the embedded systems domain. The adaptability is achieved in a controlled, formal and systematic way. Therefore, the process can be used in the context of critical and dependable systems, such as embedded systems. The two major problems related to the software development for embedded systems, namely specification and verification, can be effectively tackled based on the approach introduced in this chapter.
One extension of this work is to diversify the repository. This can be done in several ways. One way is to have several models for a domain and even several domains. Examples of other possible domains are communication systems, and integrated circuits. Another way is to put in the repository other kinds of artifacts like code segments, and other modeling formalisms. Furthermore, it is also needed to be able to derive source code from the CPN models. One possible idea to do this is to associate pieces of code (in C or Java) to the transitions of the models. Also, distributed repositories can be considered.

10. References

[Cheng et al., 1997] Cheng, A., Christensen, S., and Mortensen, K. H. (1997). Model checking Coloured Petri Nets exploiting strongly connected components. Technical report, Computer Science Department, Aarhus University, Aarhus (Denmark).

[Christensen and Haagh, 1996] Christensen, S. and Haagh, T. B. (1996). Design/CPN Overview of CPN ML Syntax. University of Aarhus, 3.0 edition.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). Model Checking. The MIT Press, Cambridge, MA, USA.

[Clarke and Wing, 1996] Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. ACM Computing Surveys, 28.

[Crnkovic, 2001] Crnkovic, I. (2001). Component-based software engineering - new challenges in software development. Software Focus, 2(4):127-133.

[Crnkovic and Grunske, 2007] Crnkovic, I. and Grunske, L. (2007). Evaluating dependability attributes of component-based specifications. In ICSE Companion, pages 157-158.

[Crnkovic et al., 2002] Crnkovic, I., Hnich, B., Jonsson, T., and Kiziltan, Z. (2002). Specification, implementation, and deployment of components. Communications of the ACM, 45(10):35-40.

[Des, 2006] Design/CPN 4.0. Meta Software Corporation and Department of Computer Science, University of Aarhus, Aarhus, Denmark. On-line version: http://www.daimi.aau.dk/ designCPN/.

[Donatelli and Thiagarajan, 2006] Donatelli, S. and Thiagarajan, P. S., editors (2006). Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006, Proceedings, volume 4024 of Lecture Notes in Computer Science. Springer.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and modal logic. In Handbook of Theoretical Computer Science, volume B, chapter 16, pages 995-1072. Elsevier Science Publisher B.V.

[Gorgônio and Perkusich, 2002] Gorgônio, K. C. and Perkusich, A. (2002). Adaptation of Coloured Petri Nets models of software artifacts for reuse. In Gacek, C., Software Reuse: Methods, Techniques and Tools. VII International Conference on Software Reuse, number 2319 in Lecture Notes in Computer Science, pages 240-254, Austin, Texas (USA). Springer-Verlag.

[Gorton et al., 2006] Gorton, I., Heineman, G. T., Crnkovic, I., Schmidt, H. W., Stafford, J. A., Szyperski, C. A., and Wallnau, K. C, editors (2006). Component-Based Software Engineering, 9th International Symposium, CBSE 2006, V"aster°as, Sweden, June 29 - July 1, 2006, Proceedings, volume 4063 of Lecture Notes in Computer Science. Springer.

[Jensen, 1992] Jensen, K. (1992). Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, volume 1 of EACTS - Monographs on Theoretical Computer Science. Springer-Verlag.

[Jensen, 1997] Jensen, K. (1997). Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, volume 2 of EACTS - Monographs on Theoretical Computer Science. Springer-Verlag.

[Jensen, 2005] Jensen, K., editor (2005). Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, volume PB-576, Aarhus (Denmark). DAIMI.

[Jensen, 2006] Jensen, K., editor (2006). Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, volume PB-579, Aarhus (Denmark). DAIMI.

[Kleijn and Yakovlev, 2007] Kleijn, J. and Yakovlev, A., editors (2007). Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings, volume 4546 of Lecture Notes in Computer Science. Springer.

[Knight, 2001] Knight, J. C. (2001). Dependability of embedded systems. In Proceedings of the 23rd International Conference on Software Engineering, page 688.4. IEEE Computer Society.

[Knight, 2002] Knight, J. C. (2002). Dependability of embedded systems. In Proceedings of the 24th International Conference on Software Engineering, pages 685-686. ACM Press.

[Knight, 2004] Knight, J. C. (2004). An introduction to computing system dependability. In Proceedings of the 26th International Conference on Software Engineering, pages 730-731. IEEE Computer Society.

[Land and Crnkovic, 2007] Land, R. and Crnkovic, I. (2007). Software systems in-house integration: Architecture, process practices, and strategy selection. Information & Software Technology, 49(5):419-444.

[Lee, 1999] Lee, E. A. (1999). Embedded software - an agenda for research. Technical Report UCB/ERL No. M99/63, University of California at Berkeley.

[Lee, 2002] Lee, E. A. (2002). Embedded software. In Zelkowitz, M., editor, Advances in Computers, volume 56. Academic Press, London (UK).

[Lemos and Perkusich, 2001] Lemos, A. J. P. and Perkusich, A. (2001). Reuse of Coloured Petri Nets software models. In Proc. of The Eighth International Conference on Software Engineering and Knowledge Engineering, SEKE'Ol, pages 145-152, Buenos Aires (Argentina).

[Long, 1993] Long, D. L. (1993). Model Checking, Abstraction, and Compositional Reasoning. PhD thesis, Carnegie Mellon University.

[McMillan, 1993] McMillan, K. L. (1993). Symbolic Model Checking. Kluwer Academic Publishers, Boston/ Dordrecht/ London.

[Murata, 1989] Murata, T. (1989). Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4):541-580.

[Nierstrasz et al., 2002] Nierstrasz, O., Arevalo, G., Ducasse, S., Wuyts, R., Black, A. P., Miiller, P. O., Zeidler, C, Genssler, T., and van den Born, R. (2002). A component model for field devices. Lectures Notes in Computer Science, 2370:200-216.

[Nierstrasz et al., 1992] Nierstrasz, O., Gibbs, S., and Tsichritzis, D. (1992). Component-oriented software development. Communications of the ACM, 35(9):160-165.

[Peled, 1994] Peled, D. (1994). Combining partial order reductions with on the fly model checking. In CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification, pages 377-390, London, UK. Springer-Verlag.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77), pages 46–57, Providence, Rhode Island. IEEE, IEEE Computer Society.

[Ramadge and Wonham, 1987] Ramadge, P. J. G. and Wonham, W. M. (1987). On the supremal controllable sublanguage of a given language. SLAM Journal on Control and Optimization, 25(3):637-659.

[Ramadge and Wonham, 1989] Ramadge, P. J. G. and Wonham, W. M. (1989). The control os discrete event systems. Proceedings of the IEEE, 77(l):81-97.

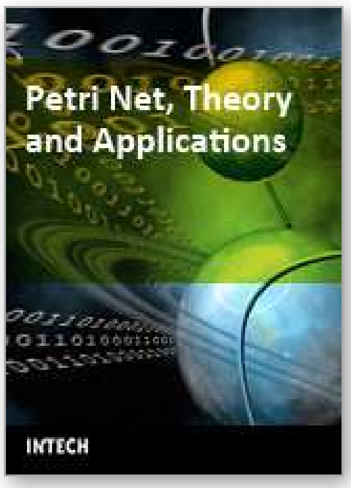
[Silva and Perkusich, 2005] Silva, L. D. and Perkusich, A. (2005). A Model-Based Approach to Formal Specification and Verification for Embedded Systems Using Coloured Petri Nets. In: Colin Atkinson; Christian Bunse; Hans-Gerhard Gross; Christian Peper. (Org.). Component-Based Software Development for Embedded Systems: An Overview on Current Research Trends. Berlin: Springer-Verlag, v. 3778, p. 35-58.

[Szyperski, 1999] Szyperski, C. (1999). Component Software: Beyond Object-Oriented Programming. Addison-Wesley.

[Ullman, 1998] Ullman, J. D. (1998). Elements of ML Programming. Prentice Hall, 2 edition.

[Valmari, 1991] Valmari, A. (1991). A stubborn attack on state explosion. In CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification, pages 156-165, London, UK. Springer-Verlag.

[van Steen et al., 1998] van Steen, M., van der Zijden, S., and Sips, H. (1998). A view on components. In Proceedings of the 9th International DEXA Workshop on Database and Expert Systems Applications, IEEE Computer Society, Los Alamitos, California.



Petri Net, Theory and Applications
Edited by Vedran Kordic

ISBN 978-3-902613-12-7
Hard cover, 534 pages
Publisher I-Tech Education and Publishing
Published online 01, February, 2008
Published in print edition February, 2008